Announcements

- Homework 3 is due as of start of class
- Programming Project 4 is assigned, due April 21st
- Midterm 2 is April 16th

Topic 7: Binary Search Trees

By: Professor Lynam

General Trees

- Definition: *General Tree*
- *A general tree T is empty or is a finite set of Nodes N such that there is a specially-designated root node r* ∈ N with the remaining N – r nodes partitioned into $m \geq 0$ *disjoint subtrees of T*
- Binary Search Trees ⊆ Binary Trees ⊆ General Trees
- Note that this is a recursive definition, meaning you can have sub-trees, and trees being forming by attaching existing trees
- A node in a general tree may have any non-negative quantity of "children"

Binary Trees

- Definition: *Binary Tree*
- *A binary tree is a general tree such that* $0 \leq m \leq 2$.

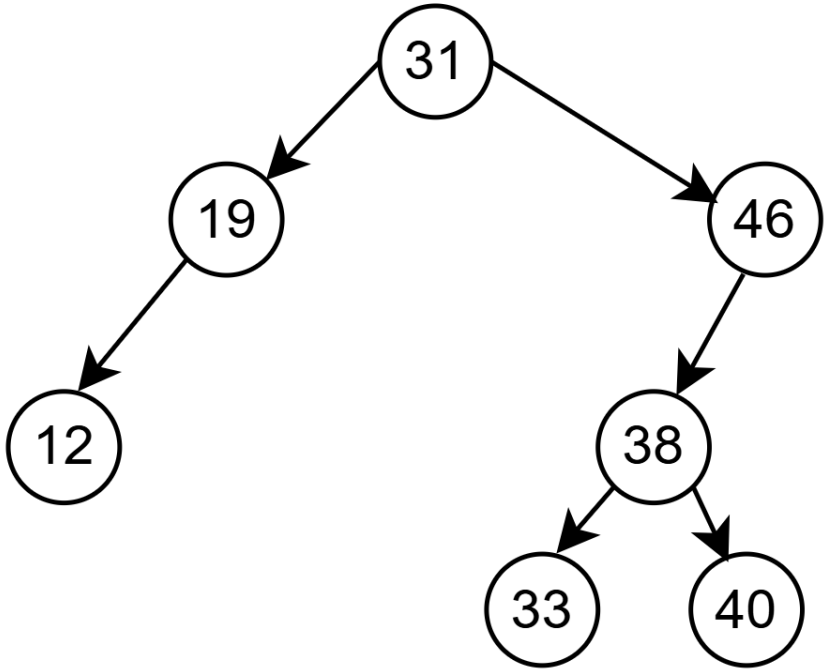
- We call the node “higher” in the tree the parent node, and assuming it has two children, it has a “left” and “right” child node
- Example applications of binary trees include:
- Binary Space Partitioning (BSP) in video games
- Huffman Coding (lossless data compression)
- Heaps (priority queues)
- Expression Trees (intermediate expression representation)
- And last, but not least…

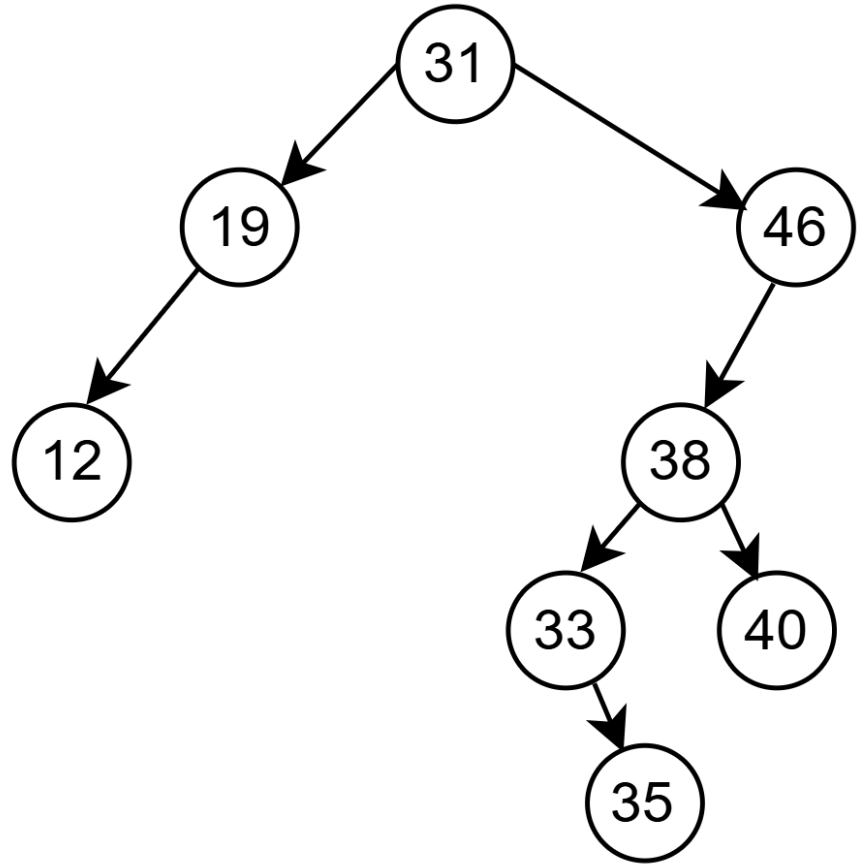
Binary Search Trees (BSTs)

- Definition: *Binary Search Tree*
- *A binary search tree (BST) is a binary tree in which any key value held in a node is greater than any key in that node's left subtree and is less than any key in its right subtree.*

- What about equal key values, you might ask?
- We can extend the definition to store ties on one side or the other (but not both!) of existing matches. (However, in many BST applications, data values are unique.)

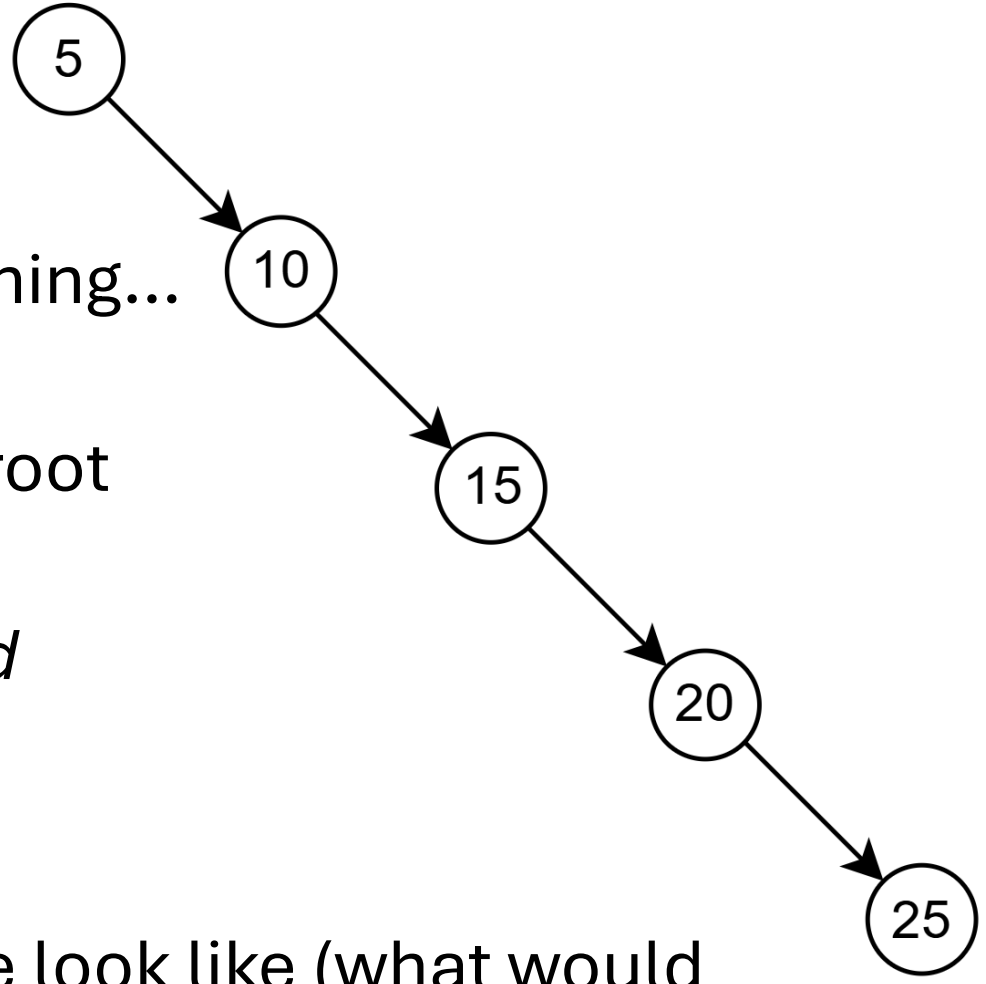
BST Insertions





- Let's insert 35 into this BST. What are the steps?
- Just search for 35, and insert into the fringe where the search fails

BST Searching



- A key advantage of BSTs is efficient searching… let's see how efficient it actually is
- Best case: O(1), when the target is in the root node
- Worst case: O(n), when the tree is *skewed*
- What does a skewed BST look like?
- A Linked List (searching an LL is O(n) too)
- What would the opposite of a skewed tree look like (what would our *ideal* tree look like?)?
- A "full" or "complete" tree, with each level filled
- Average case: depends on the tree shape! Could be O(*h*) to O(n/2)

BST Searching

- Definition: *Complete Tree*
- *A complete tree of height h, where the root is h = 0, has all positions on all levels filled with data.*

- So, what's the capacity of a height *h* binary tree?

Height	Capacity
0	1
1	3 (1 + 2)
2	7 (3 + 4)
3	15 (7 + 8)
…	…
h	$2^{h+1} - 1 (= n)$

BST Searching

- The height of the BST correlates with the number of comparisons necessary for searching… So what is *h* in terms of *n* (the quantity of data)?
- $2^{h+1} - 1 = n$
- $2^{h+1} = n + 1$
- $\log_2 2^{h+1} = \log_2(n+1)$
- $h = \log_2(n+1) - 1$
- So h is O($\log_2 n$)
- But most BST aren't complete. Suppose one is just half-full… All we'd have to do is replace n with n/2 in our equation. We'd still get a $\log_2 n$ term

Tree Sort

- The idea for tree sort is that an inorder traversal of a BST visits the keys in ascending order
- The algorithm outline is simple:

1. Build a BST
2. ‘Inorderly’ traverse the BST

Tree Sort

- Analysis time! What's our best case scenario?
- The tree will end up being complete!

1. Build a BST: n insertions at $O(\log_2 n)$ each -> $O(n * \log_2 n)$
2. 'Inorderly' traverse the BST: T(n) = T(n/2) + 1 + T(n/2)… which is?

- O(n) by the master theorem! So overall, $O(n * \log_2 n)$
- Worst case? We get a completely skewed tree!

1. Build a BST: 1 + 2 + 3 + … + n -> $O(n^2)$
2. Traverse: still O(n)

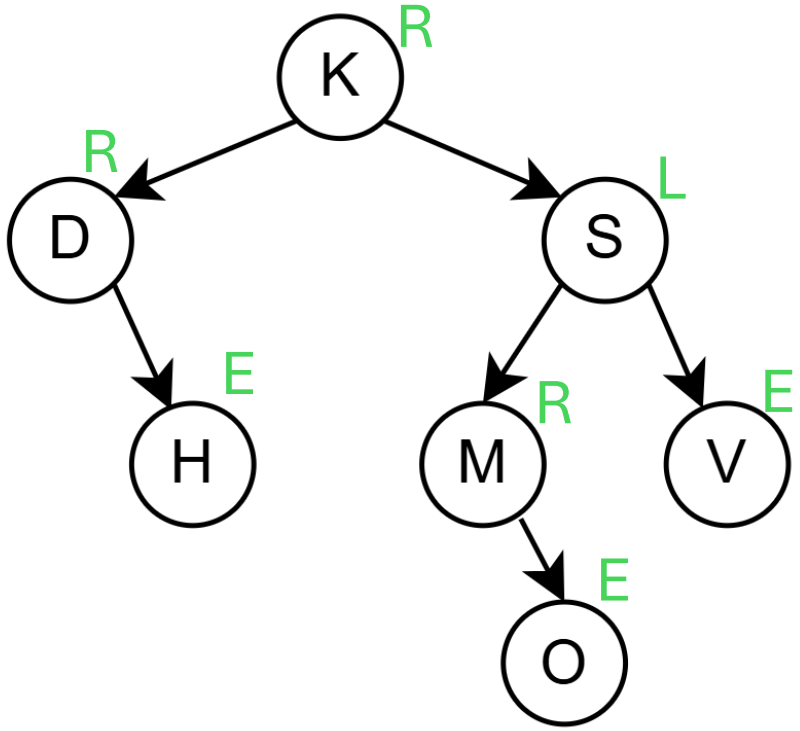
- Overall, $O(n^2)$
- Average case? Same as best case: $O(n * \log_2 n)$

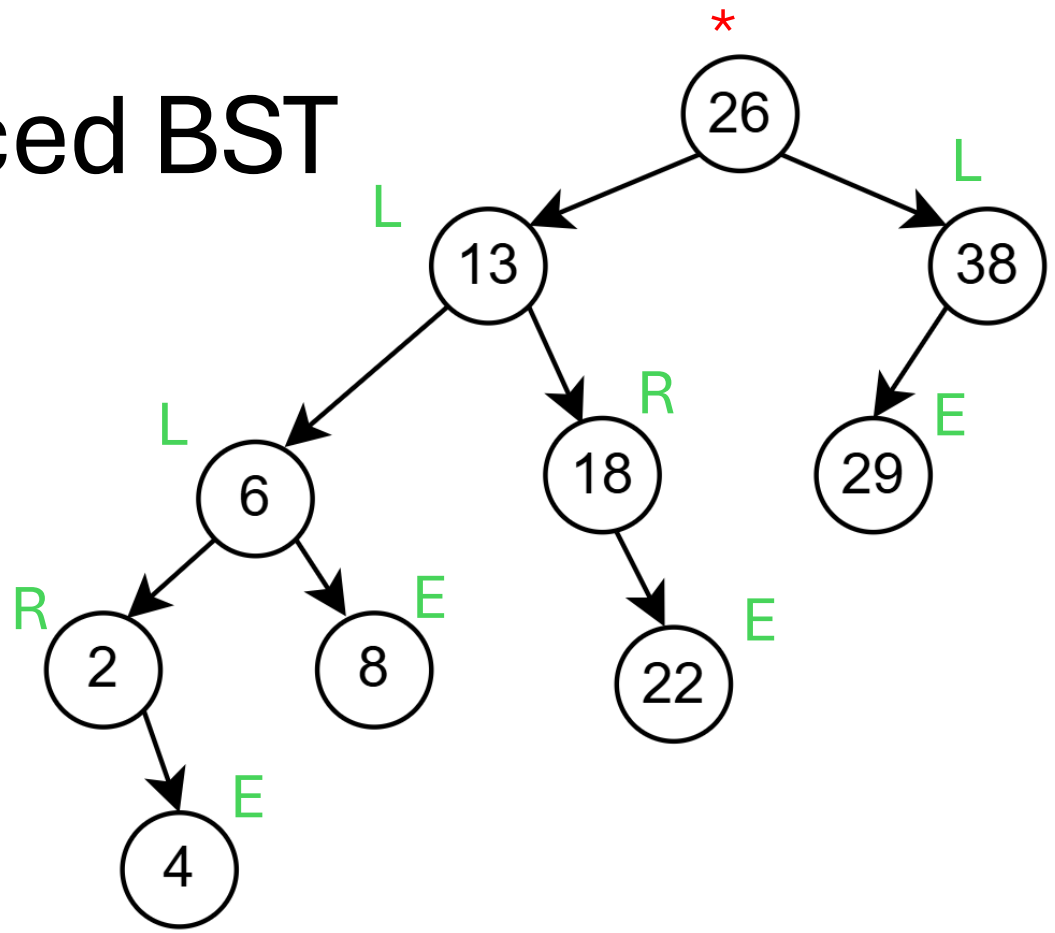
AVL Trees: A Height-Balanced BST

- Definition: *AVL Tree*
- *An AVL tree is a BST satisfying this balance condition: For each node in the tree, the height of its left subtree differs from the height of its right subtree by no more than one.*

- The name is for “Adelson-Velsky and Landis,” who invented them in 1962
- If this is a height-balanced BST, do weight-balanced trees exist?
- Yes! Huffman coding trees are one such example.

AVL Trees: A Height-Balanced BST





- E: subtree is evenly balanced
- L: subtree is left-heavy by one level
- R: subtree is right-heavy by one level
- *: marks the *pivot*, the root of the deepest unbalanced subtree

AVL Trees Insertions

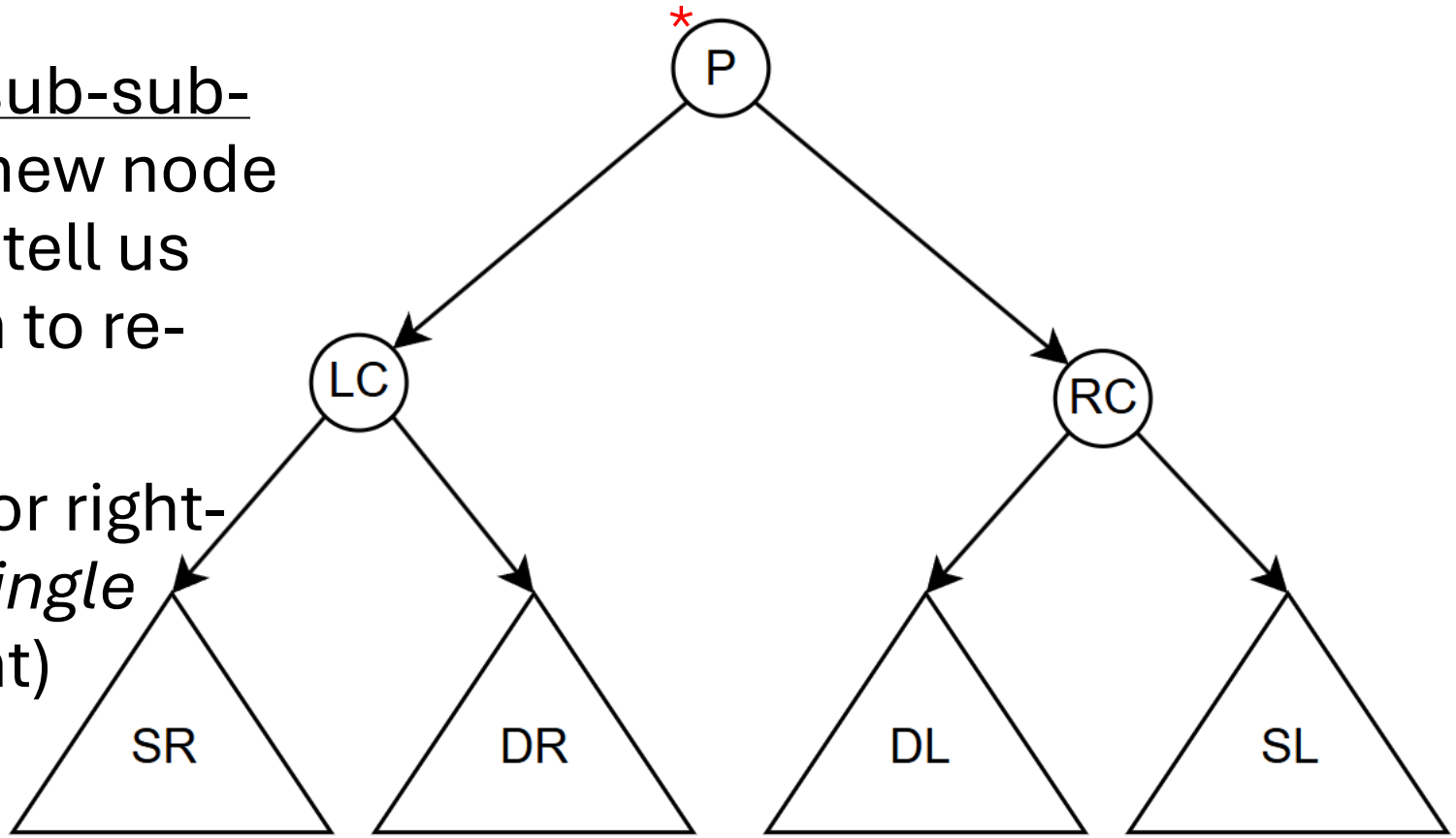
- Just two steps!

1. Perform a normal BST insertion.
2. (if necessary) Rebalance the tree.

- Done! Easy.

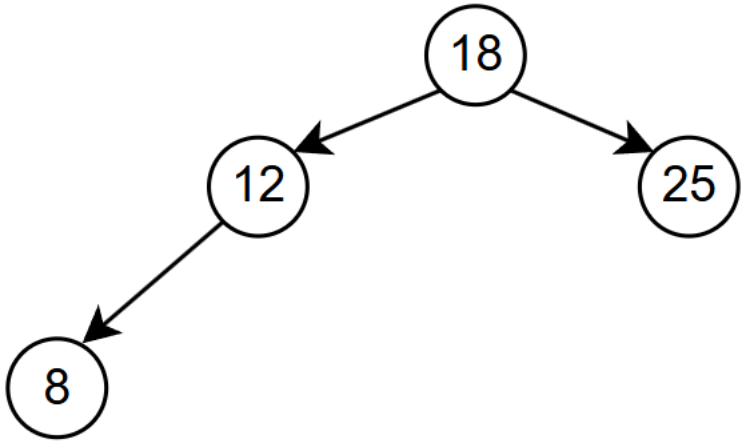
AVL Trees Rebalancing

- We need to know which sub-sub-tree of the pivot was the new node inserted. The answer will tell us which *rotation* to perform to re-balance the tree
- Insertions to the left-left or right-right subtrees require a *single rotation* (either left or right)
- Insertions to the left-right or right-left subtrees require a *double rotation* (again, either left or right)



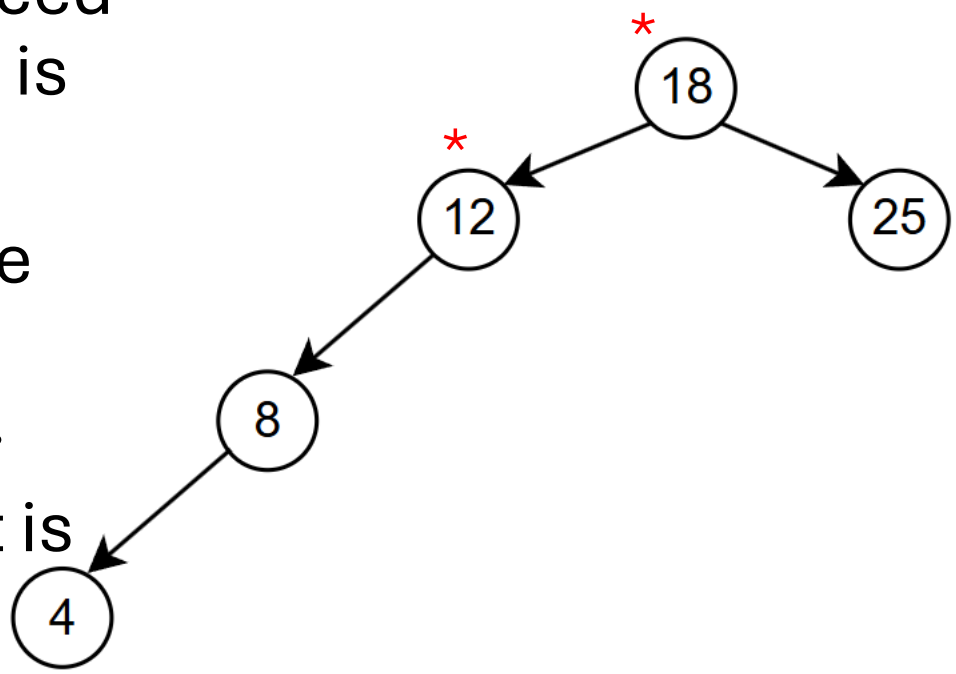
AVL Trees Insertions

- Insert 4 into this AVL tree:
- First, is it an AVL tree? What is each nodes balance?



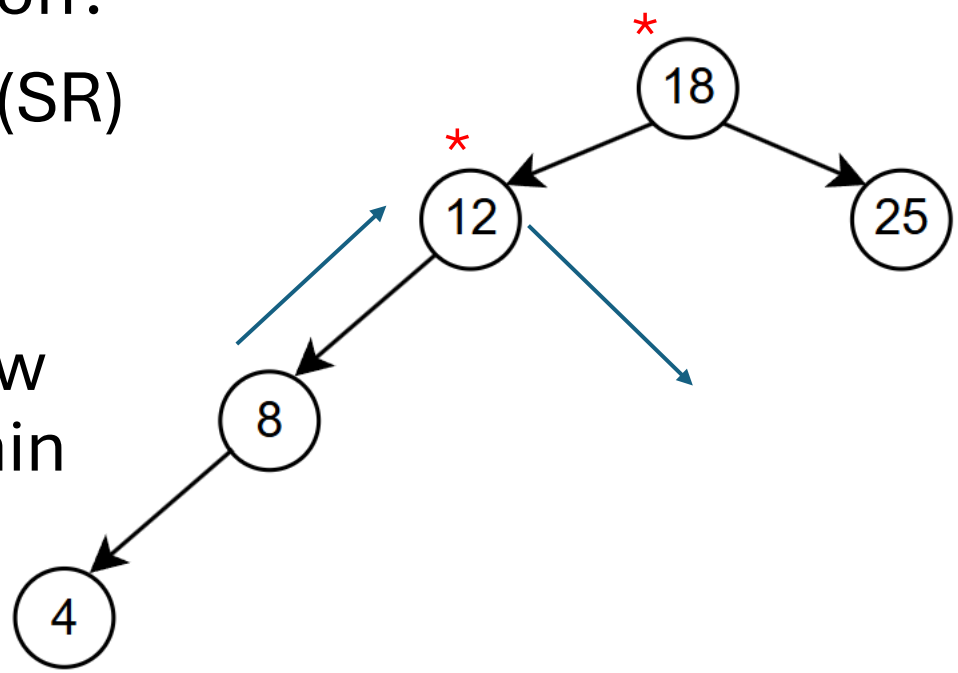
AVL Trees Insertions

- Insert 4 into this AVL tree:
- Is it unbalanced now? Where is the pivot?
- 12 and 18 are both unbalanced!
- But the pivot is the *deepest* unbalanced subtree root



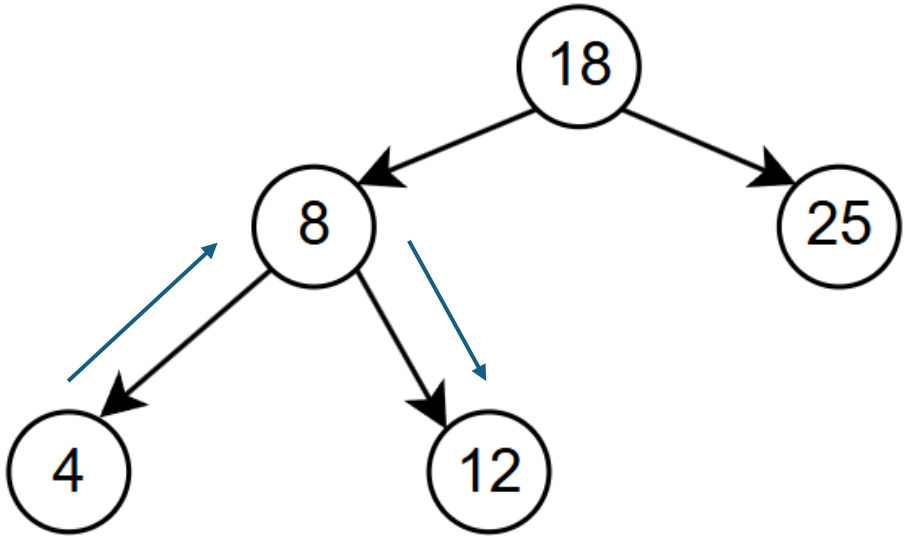
AVL Trees Insertions

- Insert 4 into this AVL tree:
- Which rotation?
- Single Right (SR)
- Imagine a snowplow pushing snow up a mountain



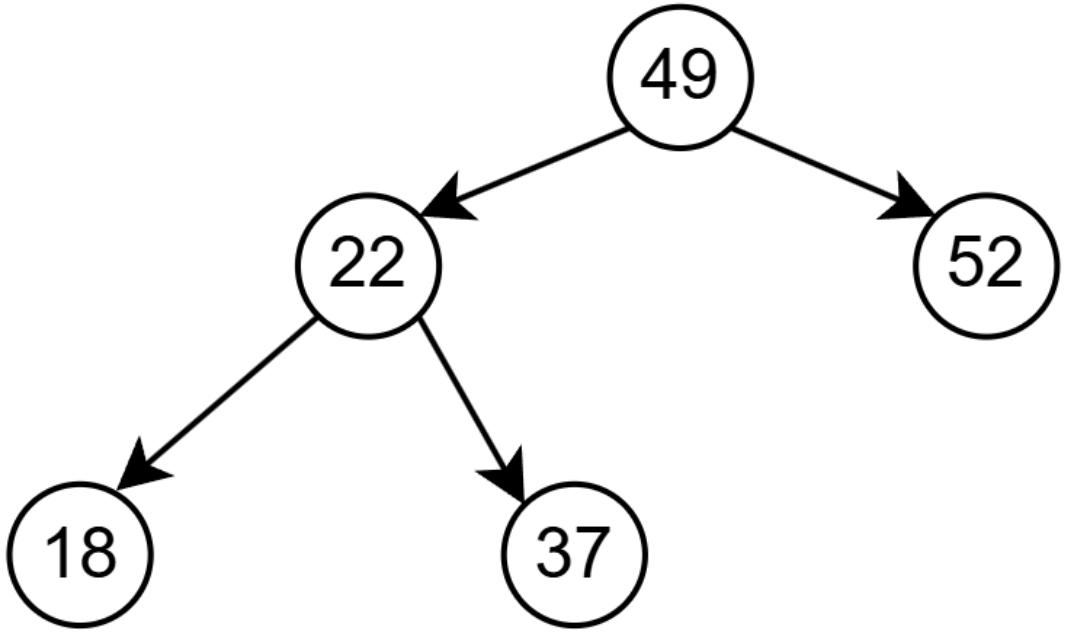
AVL Trees Insertions

- Insert 4 into this AVL tree:
- Which rotation?
- Single Right (SR)
- Imagine a snowplow pushing snow up a mountain



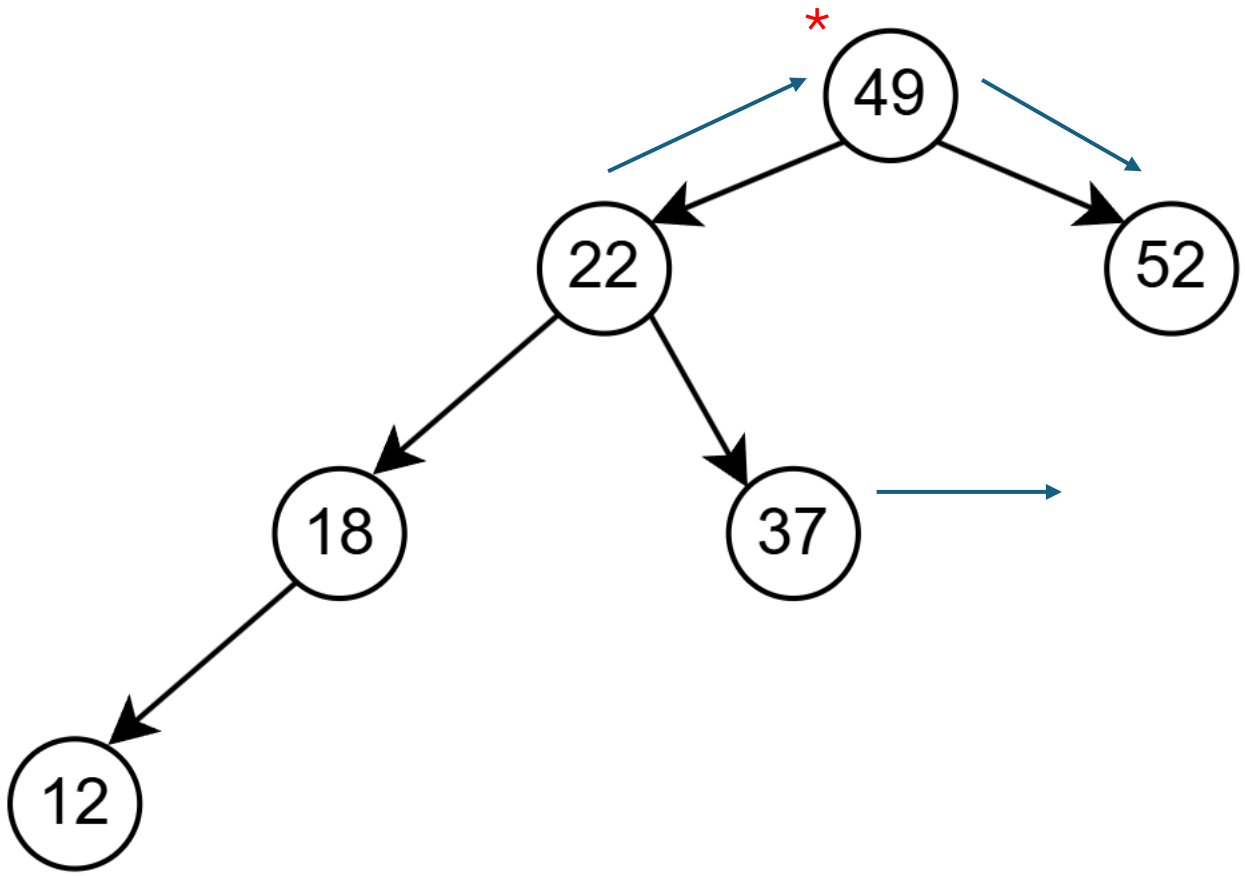
AVL Trees Insertions

- Insert 12 into this AVL tree:
- First, is it an AVL tree? What is the balance of each node?
- Where would 12 go? Would it cause it to become unbalanced?



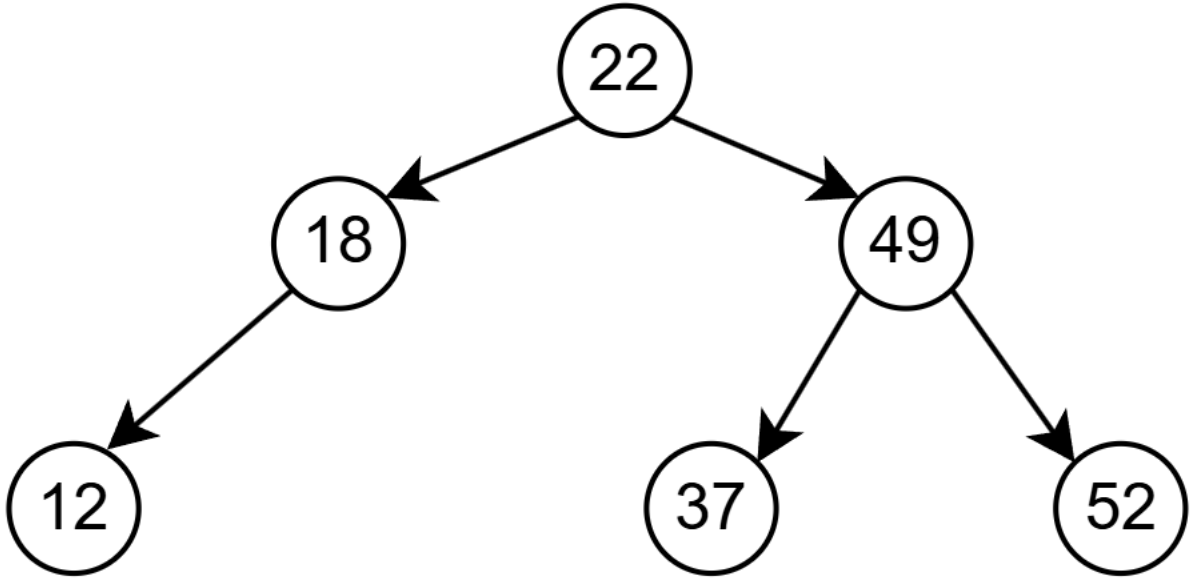
AVL Trees Insertions

- Insert 12 into this AVL tree:
- What would the pivot be?
  What would the rotation be?



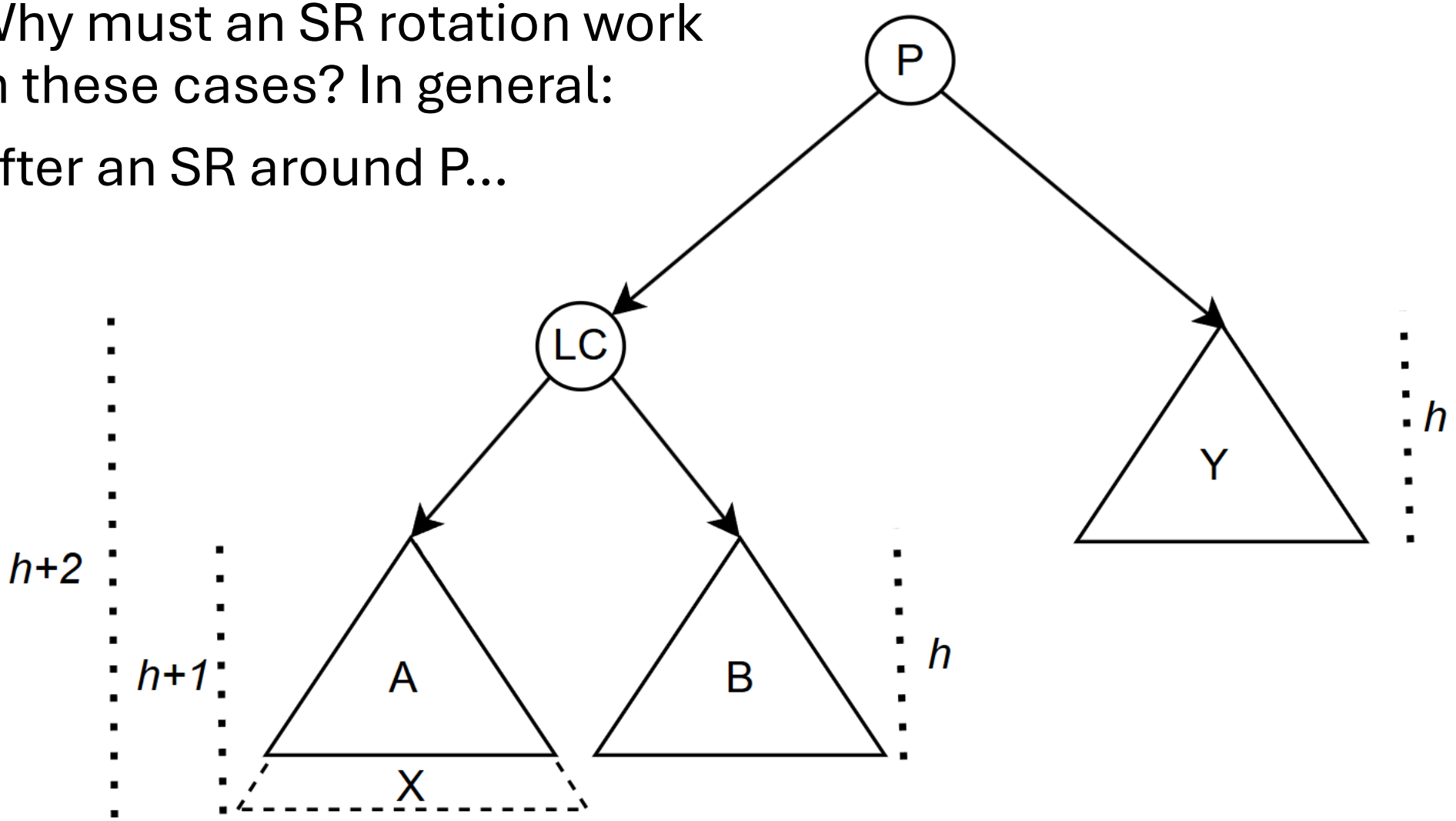
AVL Trees Insertions

- Insert 12 into this AVL tree:
- Another SR, with the right subtree of 22 becoming the left subtree of 49
- There will always be a place for every subtree in the rotated tree



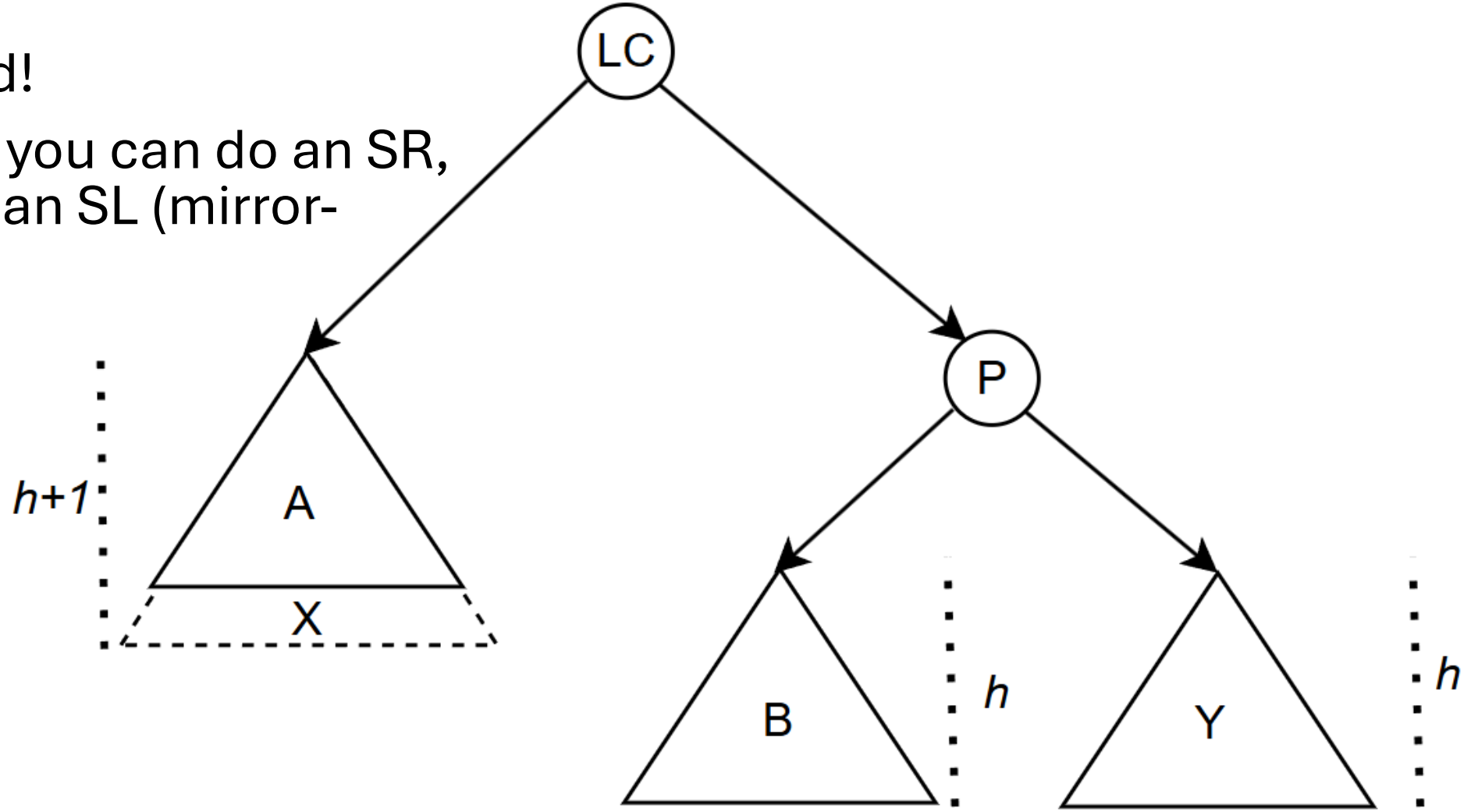
AVL Trees Single Rotations

- Why must an SR rotation work in these cases? In general:
- After an SR around P…



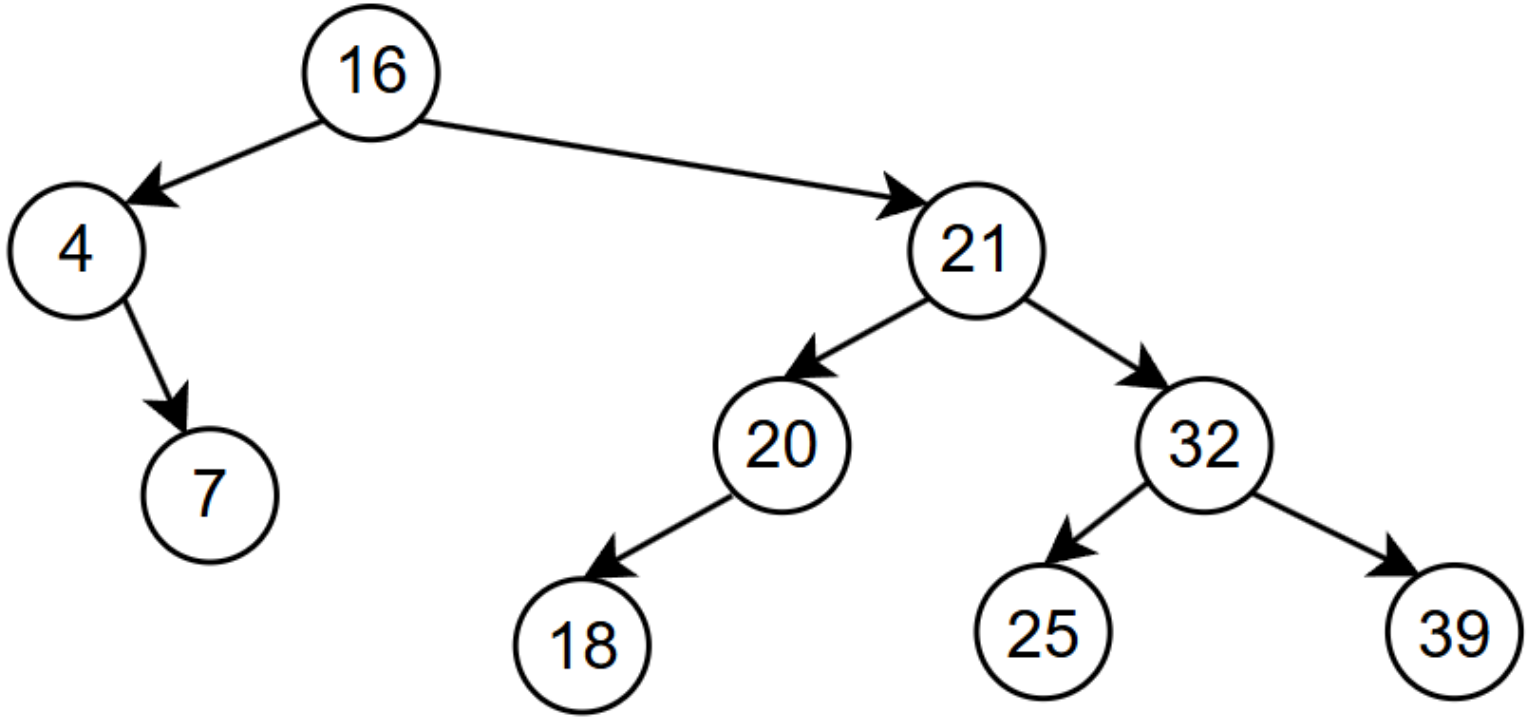
AVL Trees Single Rotations

- Rebalanced!
- Note that if you can do an SR, you can do an SL (mirror-image!)



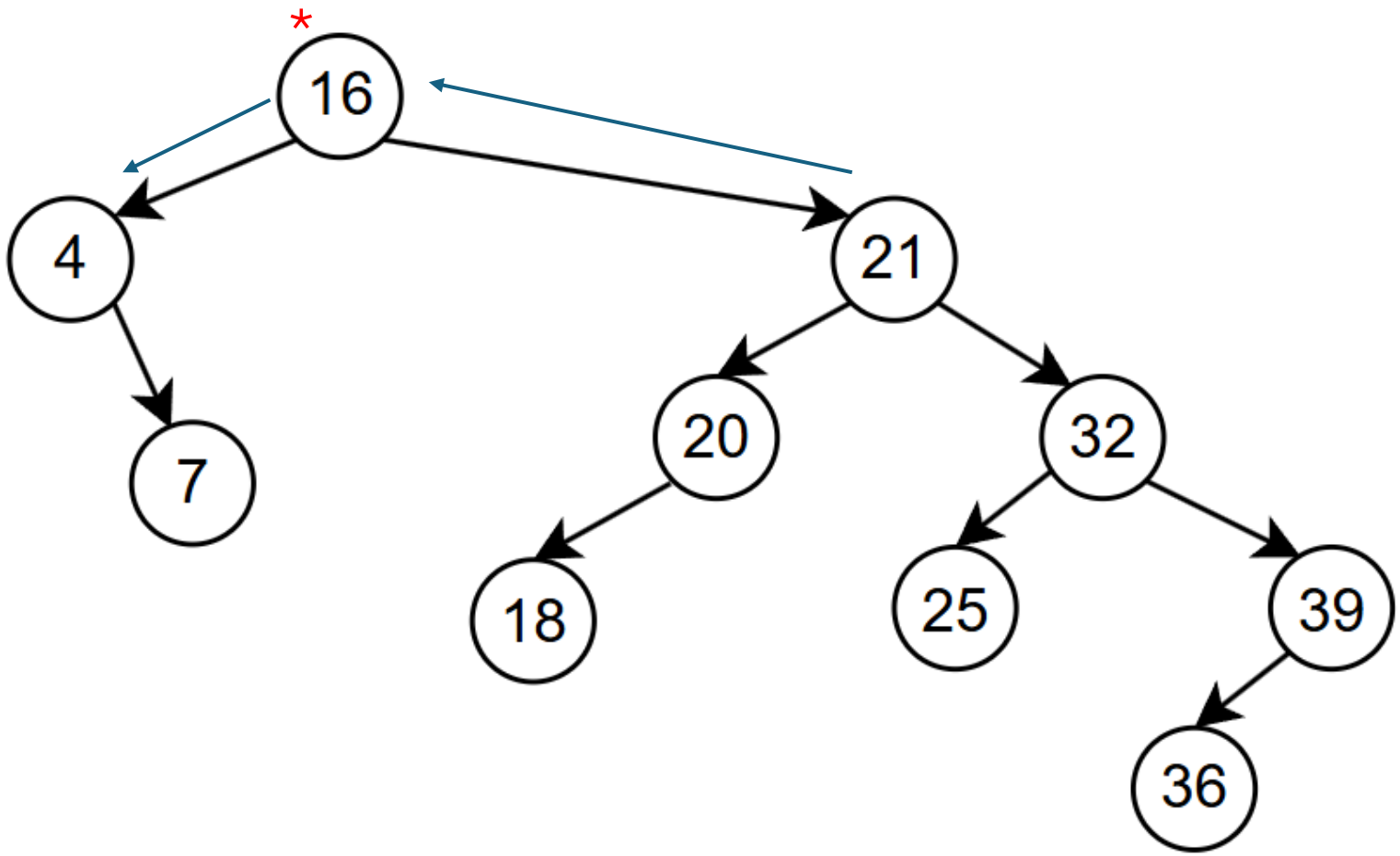
AVL Trees Insertions

- Insert 36 to this AVL tree.



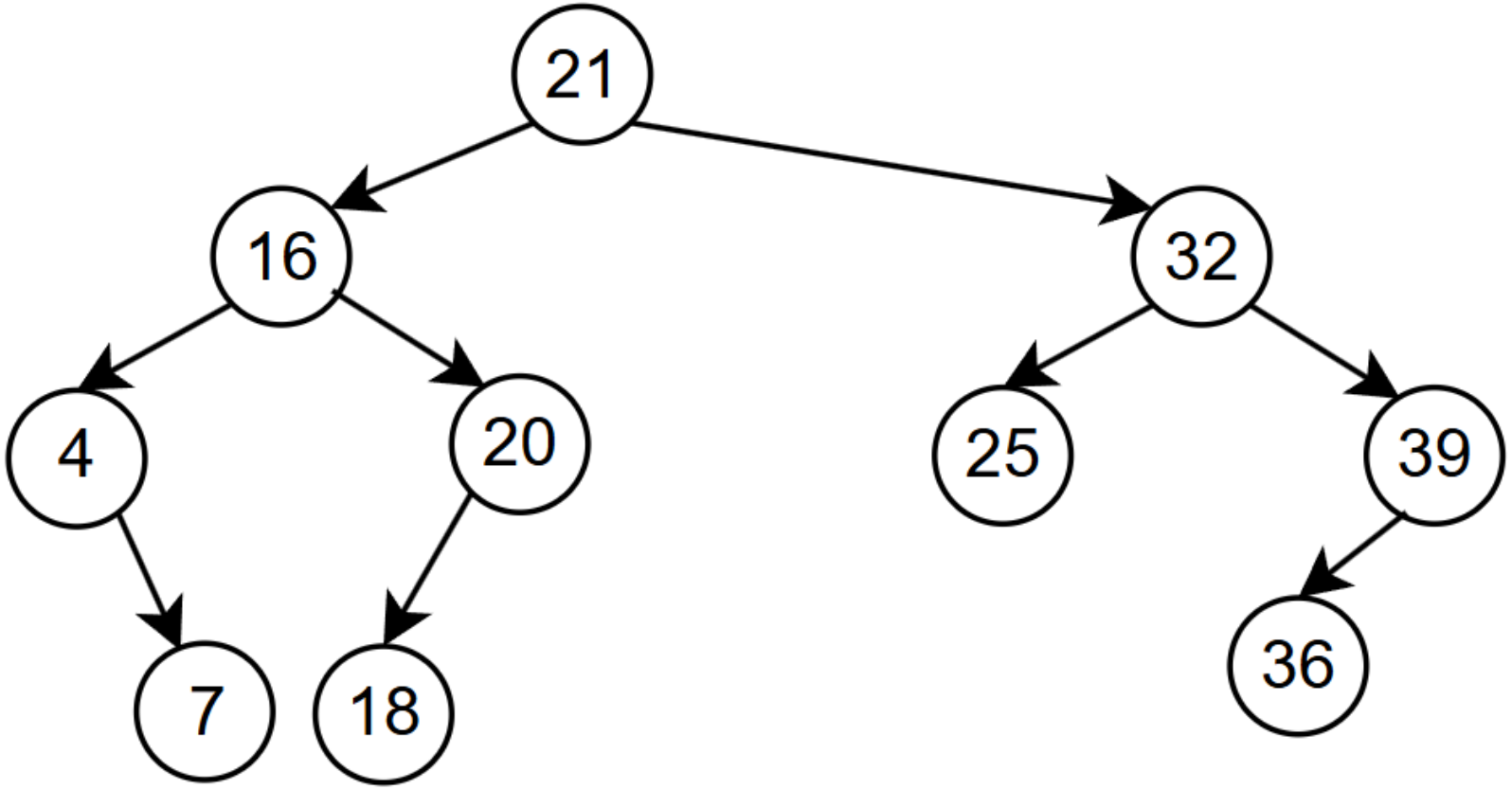
AVL Trees Insertions

- Insert 36 to this AVL tree.



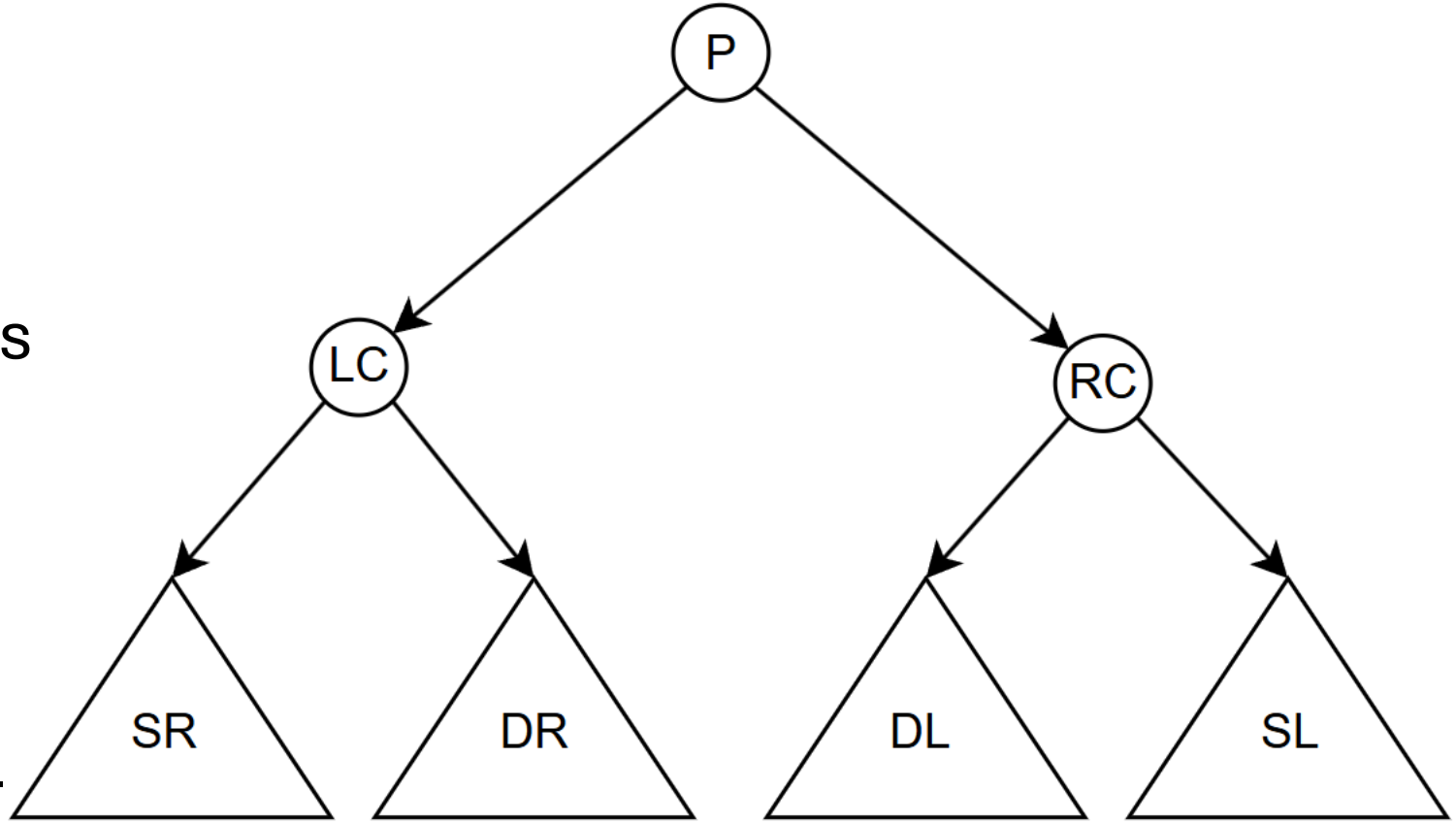
AVL Trees Insertions

- Insert 36 to this AVL tree.



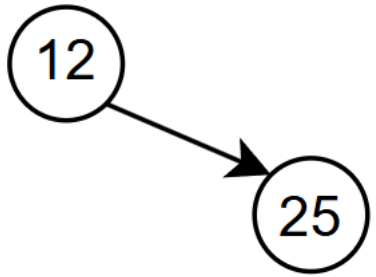
AVL Trees Double-Rotations

- Recall our pivot diagram:
- When insertions add depth to the inner sub-sub-trees, single rotations won't rebalance the tree
- We'll need *two* single rotations… aka a *double* rotation
- The result is the sub-sub-tree's root replacing the pivot (and that root is where the rotations start)



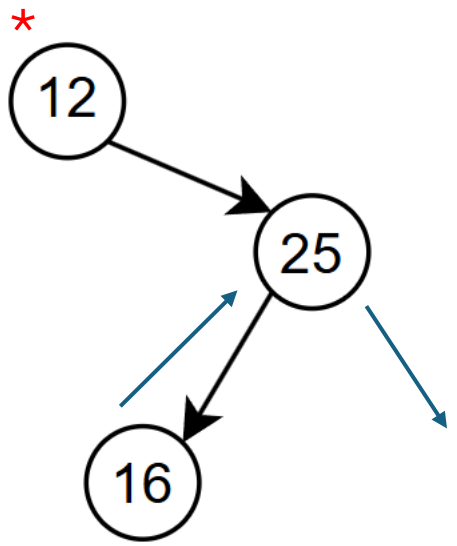
AVL Trees Double Rotations

- Insert 16 to this AVL tree.



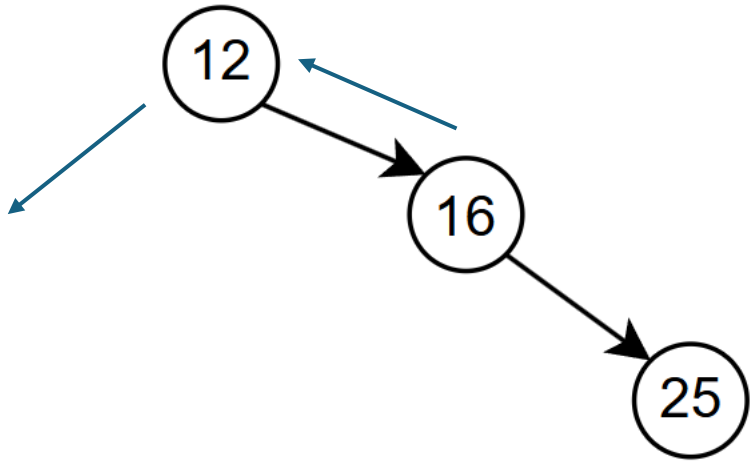
AVL Trees Double Rotations

- Insert 16 to this AVL tree.
- The goal is to get the sub-sub-tree's root two levels higher. Just follow the tree's structure to know which rotations to perform (and in which order!)
- Also note: one single rotation can't rebalance the tree: try it!



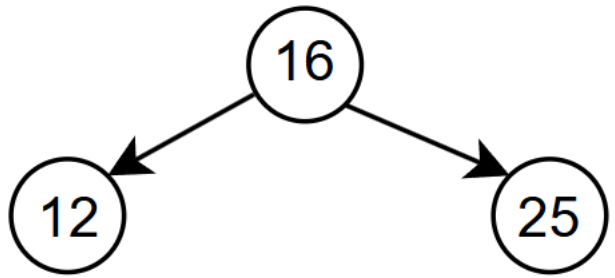
AVL Trees Double Rotations

- Insert 16 to this AVL tree.
- Start with an SR
- End with an SL



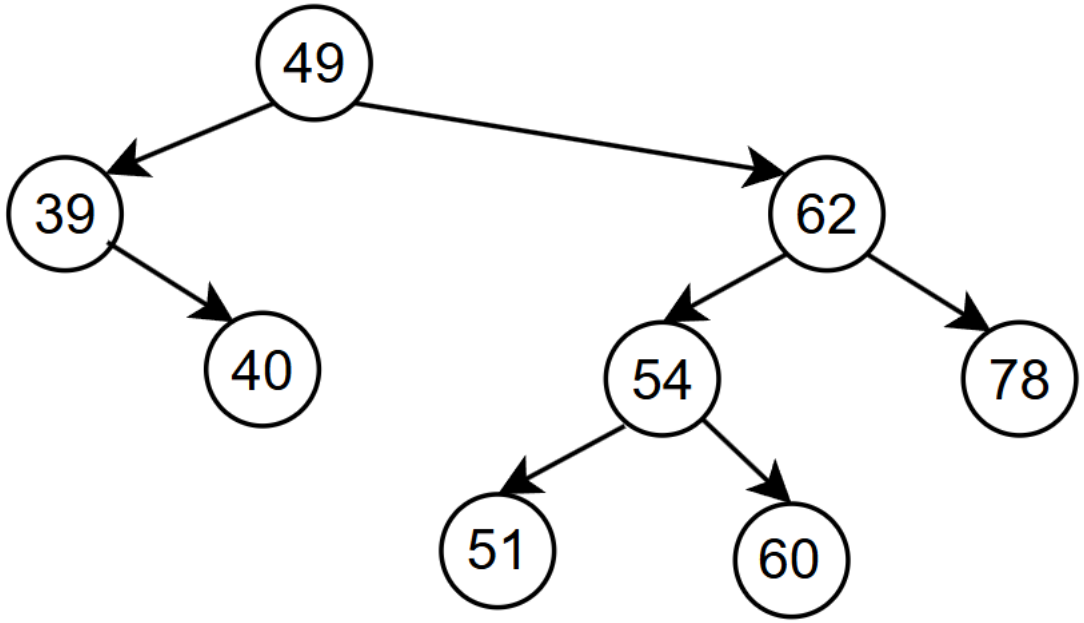
AVL Trees Double Rotations

- Insert 16 to this AVL tree.
- End with an SL



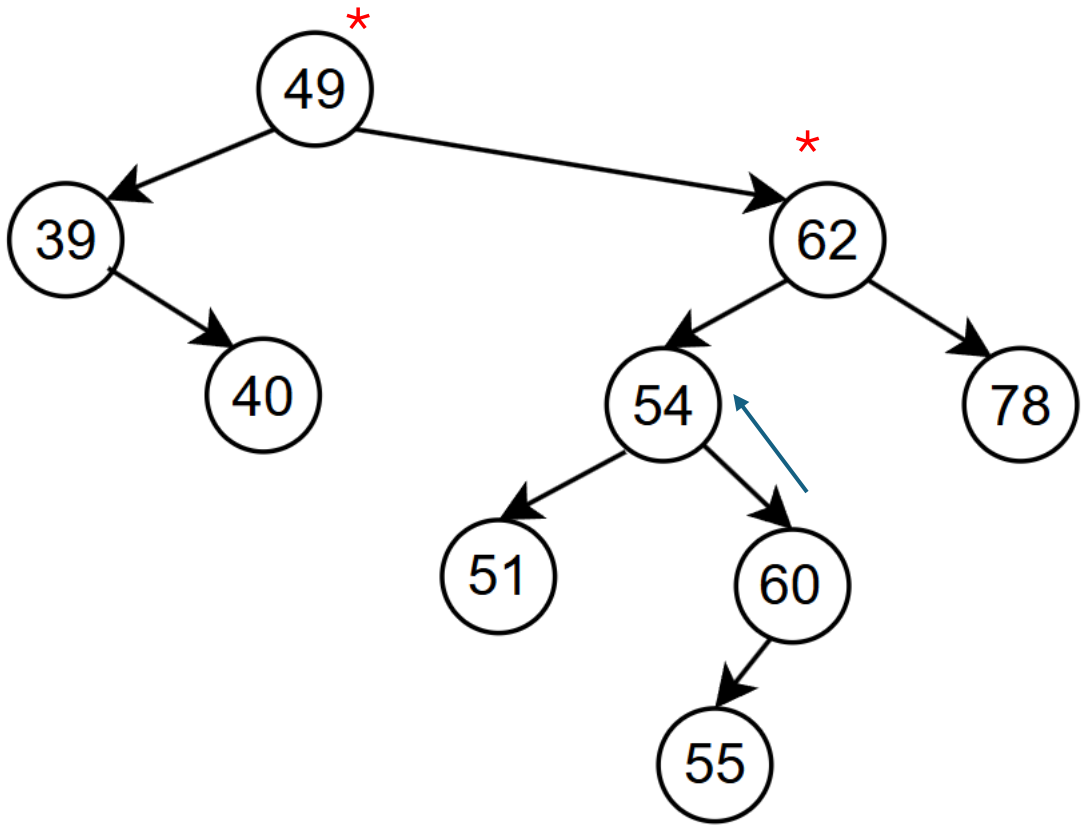
AVL Trees Double Rotations

- Insert 55 to this AVL tree.



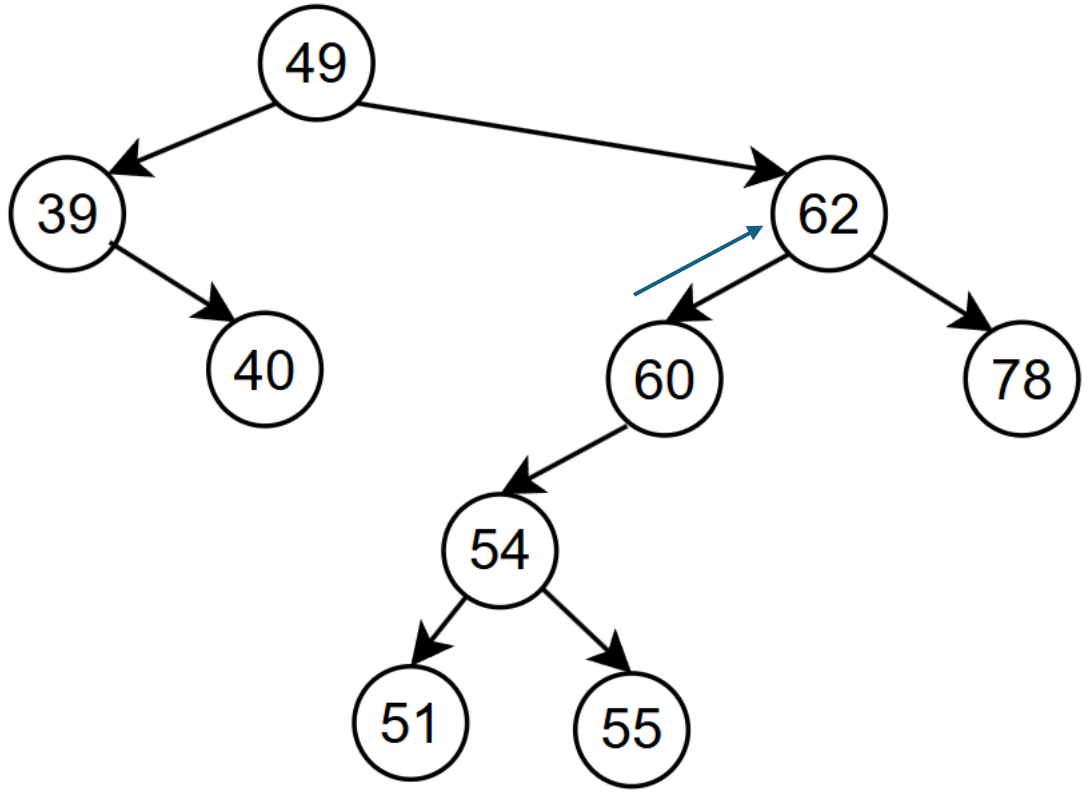
AVL Trees Double Rotations

- Insert 55 to this AVL tree.
- First, SL



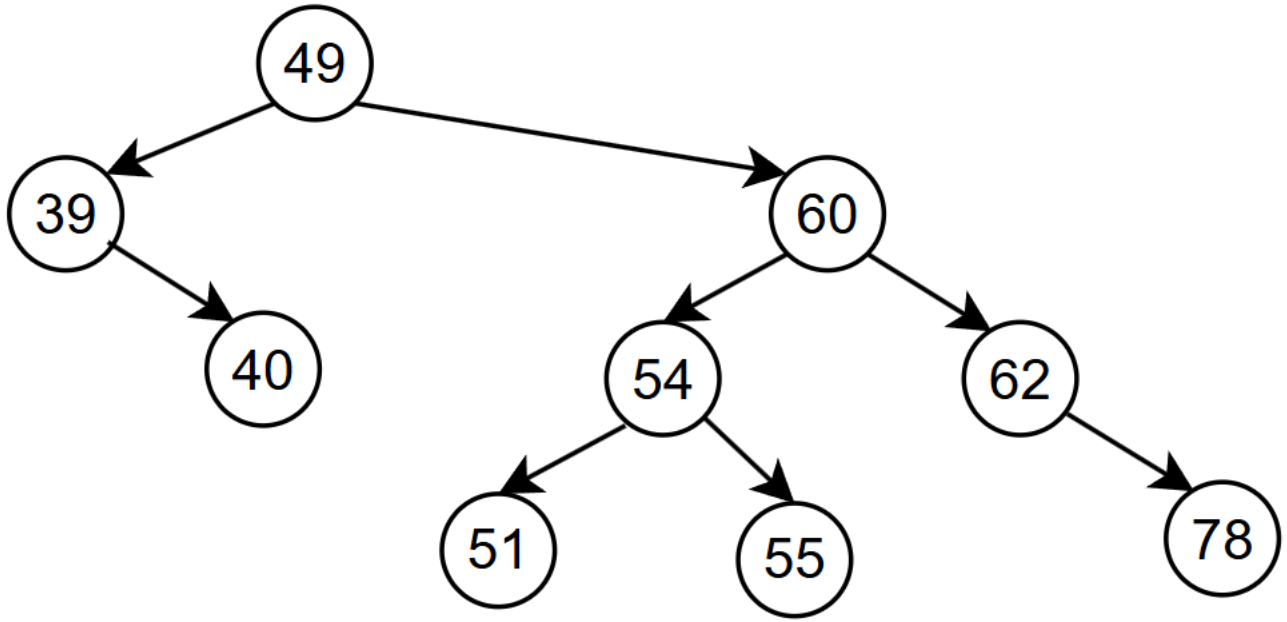
AVL Trees Double Rotations

- Insert 55 to this AVL tree.
- Now, SR



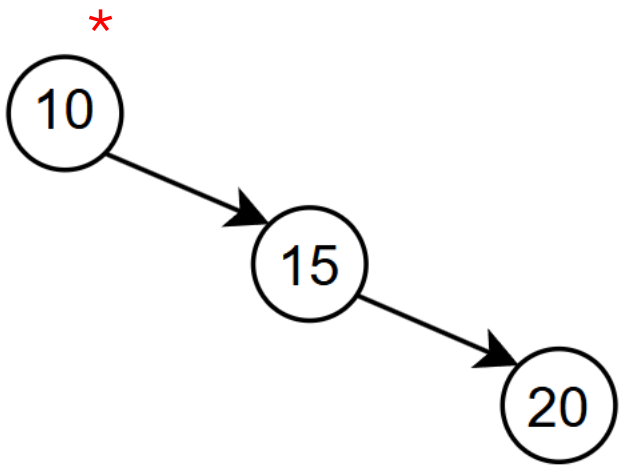
AVL Trees Double Rotations

- Insert 55 to this AVL tree.
- Note that reducing the height of the pivot fixes that whole subtree!



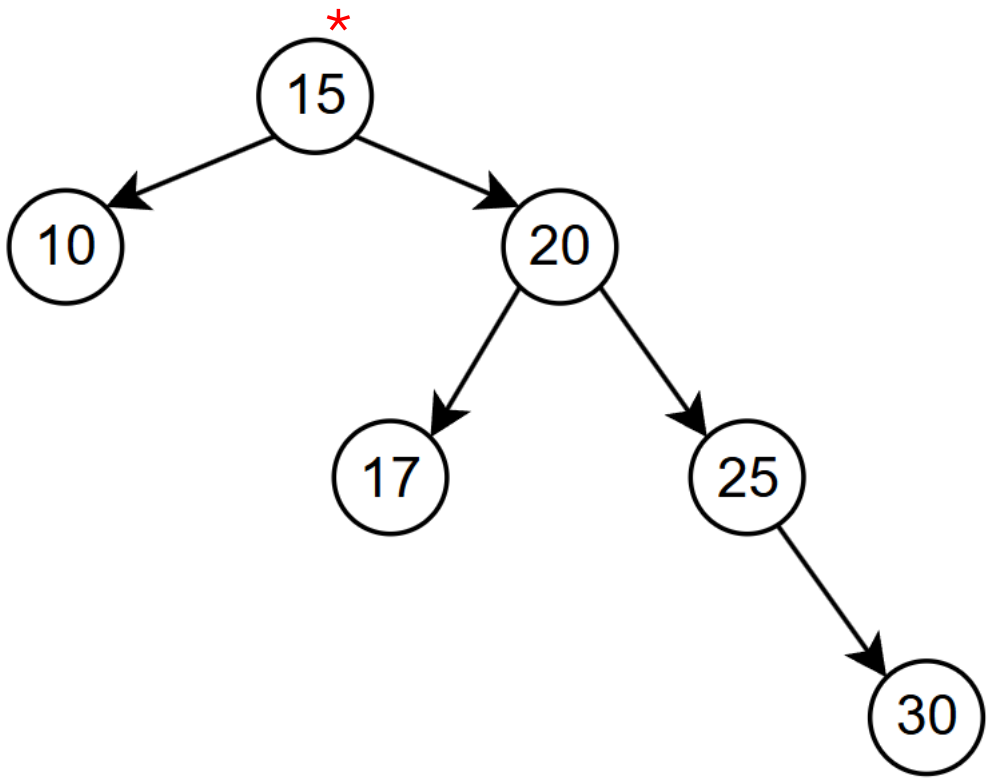
AVL Trees Building a Tree from Scratch

- Insert in order: 10, 15, 20, 25, 17, 30, 28, 27



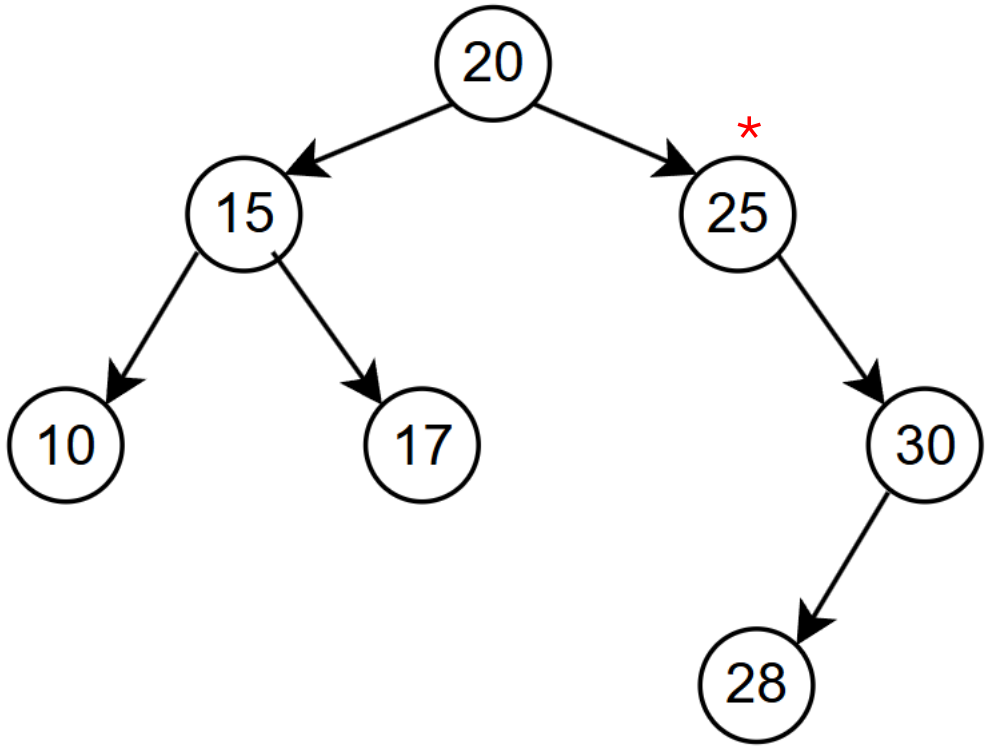
AVL Trees Building a Tree from Scratch

- Insert in order: 10, 15, 20, 25, 17, 30, 28, 27
- After an SL, keep inserting, until…



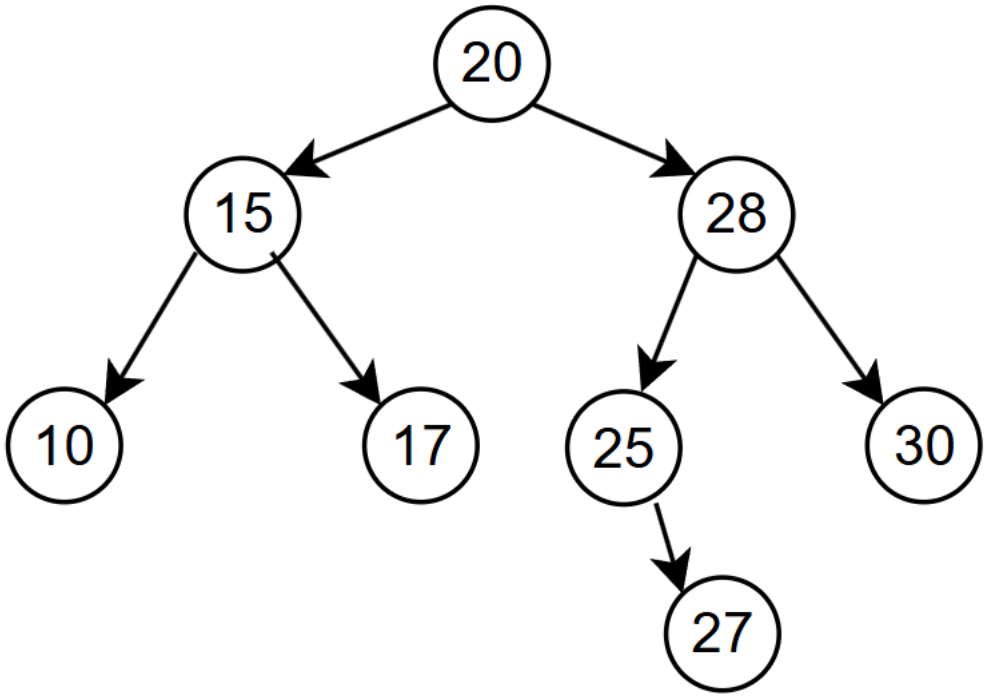
AVL Trees Building a Tree from Scratch

- Insert in order: 10, 15, 20, 25, 17, 30, 28, 27
- After another SL, keep inserting, until…



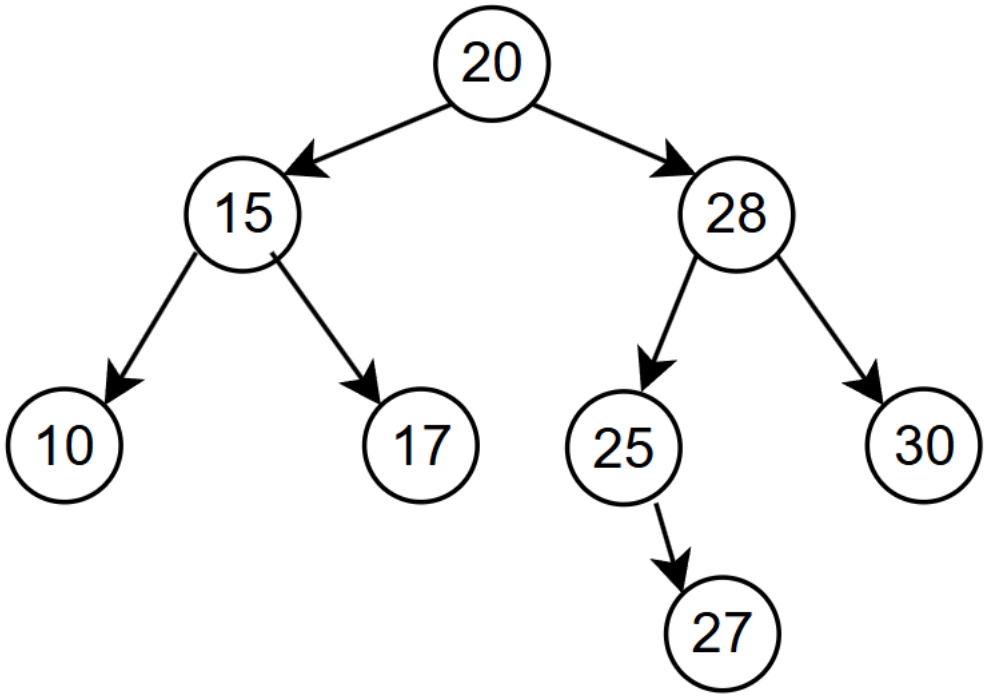
AVL Trees Building a Tree from Scratch

- Insert in order: 10, 15, 20, 25, 17, 30, 28, 27
- After a DL… Done!



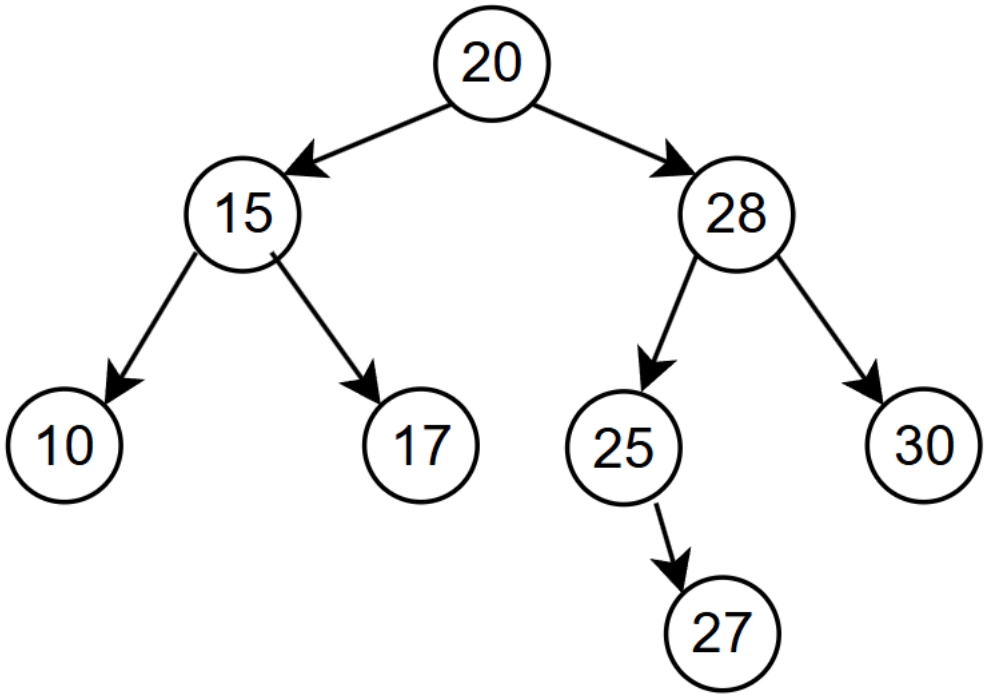
AVL Trees Deletion

- Let's back up: how do we delete from a Binary Search Tree? Suppose we wanted to delete 28.
- Just replace the target with either the inorder predecessor or inorder successor
- This turns internal deletion into leaf deletion



AVL Trees Deletion

- What about deletion from an AVL tree?
- We can do it (just start with a BST deletion)
- We can do it efficiently (involves rotations just like for insertions)
- …But we aren’t going to do it in this course



AVL Trees Performance

- Does all that work we had to do for rotations actually pay off? The resulting trees seem pretty well balanced, but are they really?
- Let's see if we can prove (I provide a proof *outline* here, skimming over a few steps) the AVL tree height is "good!"
- Conjecture: The height of an AVL tree is $O(\log_2 n)$, where n is the quantity of nodes in the tree.
- Let nodes(h) be the minimal quantity of nodes that an AVL tree of height h must have.
- A single-node tree: h=0, nodes(h) = 1
- A two-node tree: h=1, nodes(h) = 2

AVL Trees Performance

- Conjecture: The height of an AVL tree is $O(\log_2 n)$, where n is the quantity of nodes in the tree.
- Observation: the # of nodes in the tree = 1 + # nodes in one subtree + # nodes in the other subtree. This applies to nodes(h), too.
- Assuming the balance is imperfect: nodes(h) = 1 + nodes(h – 1) + nodes(h – 2).
- nodes(h – 1) > nodes(h – 2)
- So: 1 + nodes(h – 1) + nodes(h – 2) > 1 + nodes(h – 2) + nodes(h - 2)

AVL Trees Performance

- Conjecture: The height of an AVL tree is $O(\log_2 n)$, where n is the quantity of nodes in the tree.
- 1 + nodes(h – 1) + nodes(h – 2) > 1 + nodes(h – 2) + nodes(h -2)
- Replacing the LHS, dropping the "+ 1," and merging, we get:
- nodes(h) > 2 * nodes(h – 2)
- This is a recurrence relation!
- Initial Condition: nodes(0) = 1
- General Form: nodes(h) $> 2^k *$ nodes(h – 2k)
- Solving for h gives us h < $2 * \log_2 n$
- Which gives us h is $O(\log_2 n)$. QED.

AVL Trees Performance

- So, if h is $O(\log_2 n)$, searching the AVL tree must be $O(\log_2 n)$.
- What about insertion?
- To complete an insertion, we need to:
- Search for the insertion location: $O(\log_2 n)$.
- Attach the key to the tree O(1)
- Locate the (potential) pivot: $O(\log_2 n)$.
- (Maybe) perform a rotation: O(1)
- Combined: $O(\log_2 n)$.
- For deletion: also $O(\log_2 n)$.

Splay Tree: A Self-Adjusting BST

- The idea is to place frequently-accessed keys in tree locations that are easily searched
- Those locations are at or near the root, but…
- There aren’t many storage locations at or near the root of a binary tree!
- If there are only some percentage of keys that are frequently accessed (as with, say, the Pareto Principle), then maybe this isn’t a big concern
- But if many keys are ‘popular,’ then the splay tree is potentially very useful
- How do we bring ‘popular’ keys up the tree?
- By ‘splaying’ them… with AVL tree rotations!

Aside: Amortized Analysis

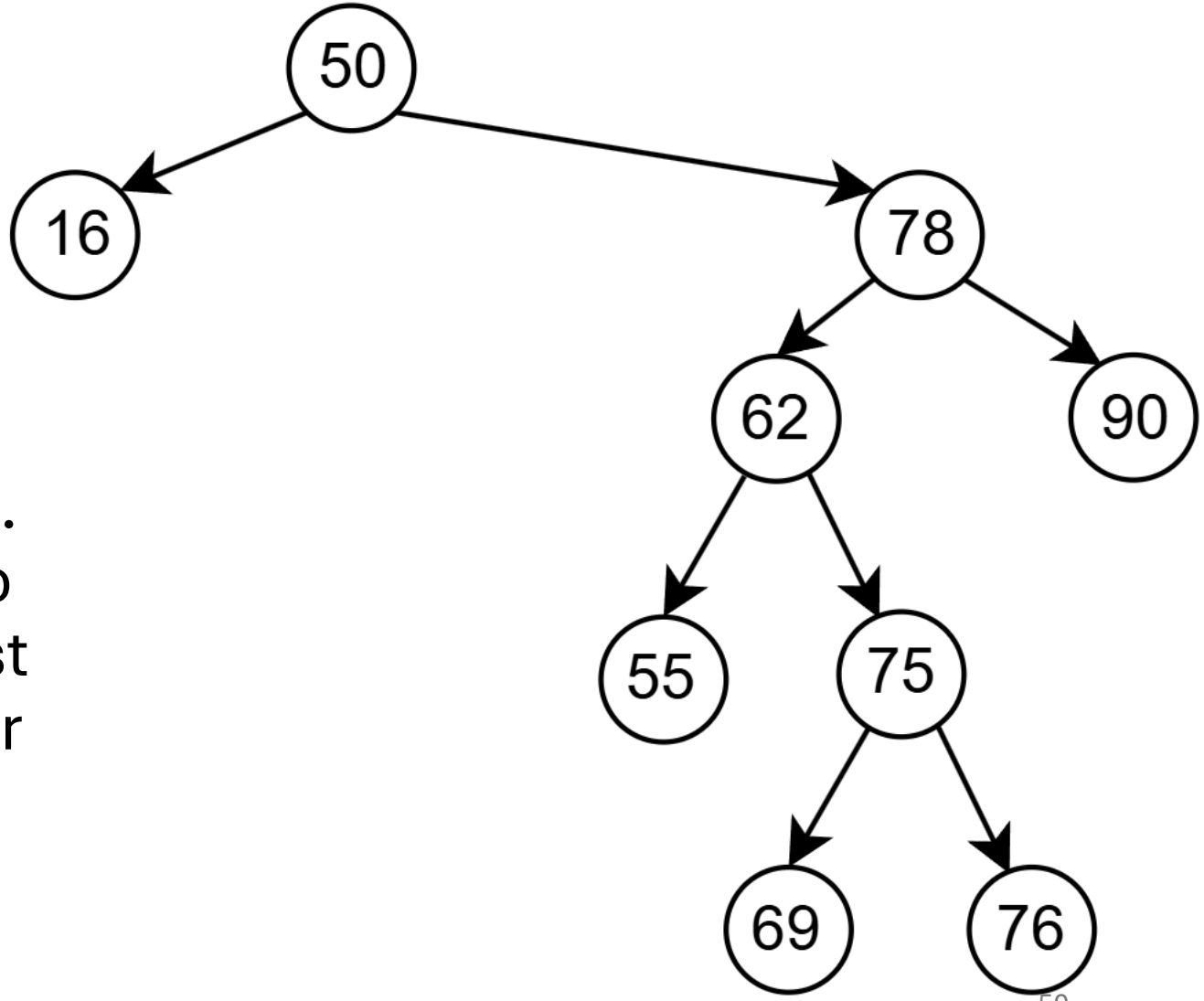
- In accounting, *amortization* is the time it takes a business to pay off a debt or an asset's cost.
- In algorithm analysis, the term is used to describe situations in which the *average* cost of an operation can be good across many executions, despite individual executions of the operation occasionally being slow.
- Why are we talking about this? Because it applies to Splay Tree operations (search and insert)

Splay Tree: A Self-Adjusting BST

- When we search for or insert a key, we do it as we would in a normal BST, but we also ‘splay’ (move) that key to the root while also restructuring the tree at the same time.
- There are three splaying operations:
- Zig: If the node that needs to be moved to the root is a child of the root, do the appropriate AVL single rotation (either SL or SR)
- Zig-Zag: Find the grandparent of the key being moved. If their positions fit an AVL double rotation, perform it.
- Zig-Zig: If neither a zig nor a zig-zag apply, perform two AVL single rotations (either SL-SL or SR-SR), the first around the grandparent and parent, the second around the parent and key being moved

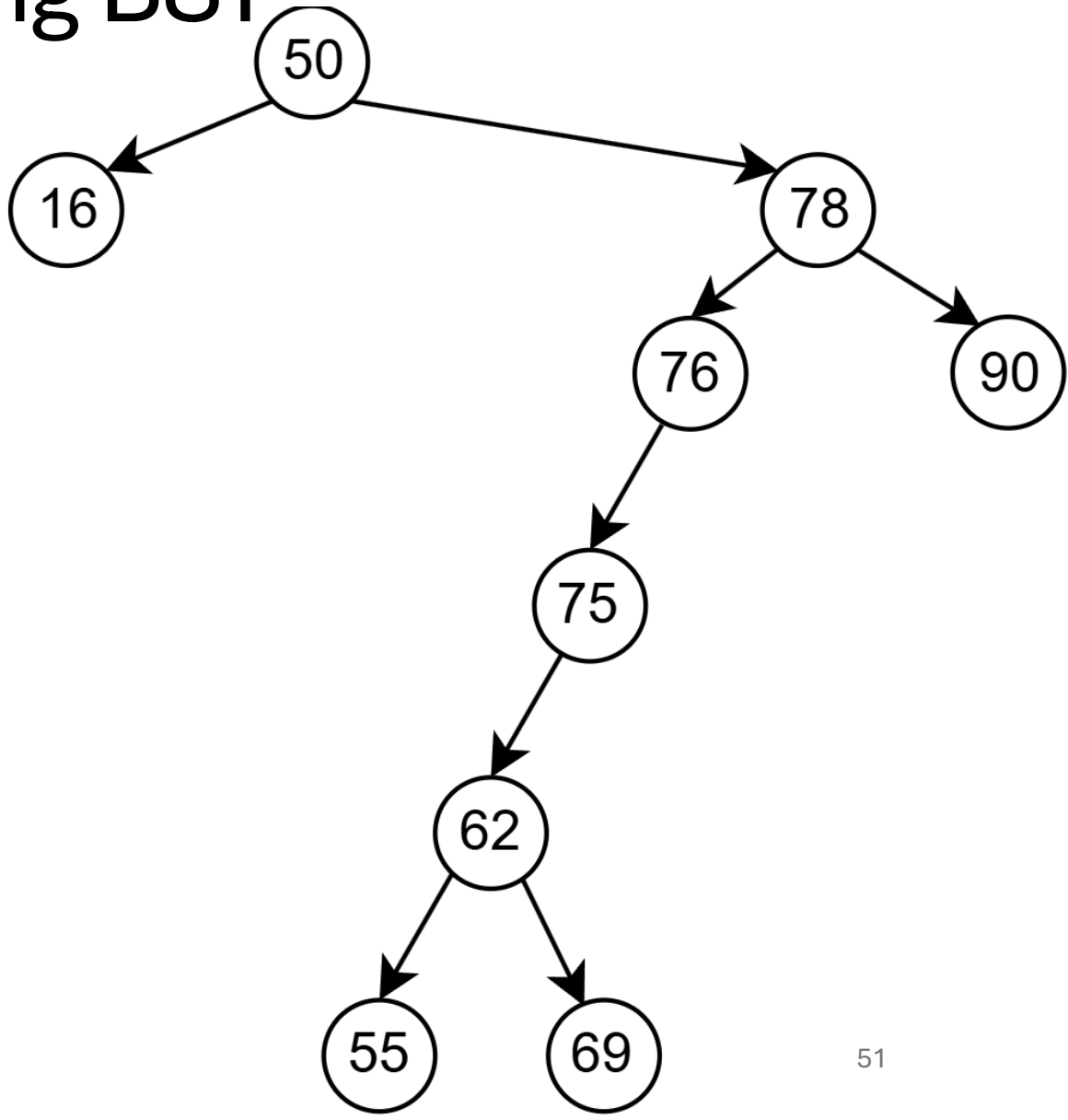
Splay Tree: A Self-Adjusting BST

- Let’s do an example! Take this BST, and search for 76. If found, splay it to the root.
- Found it! It’s below 75. Now what?
- We need to splay it to the root. We can move up (at most) two levels at a time. Would the first movement be a Zig, Zig-Zag, or Zig-Zig?
- Zig-Zig! Two SL rotations



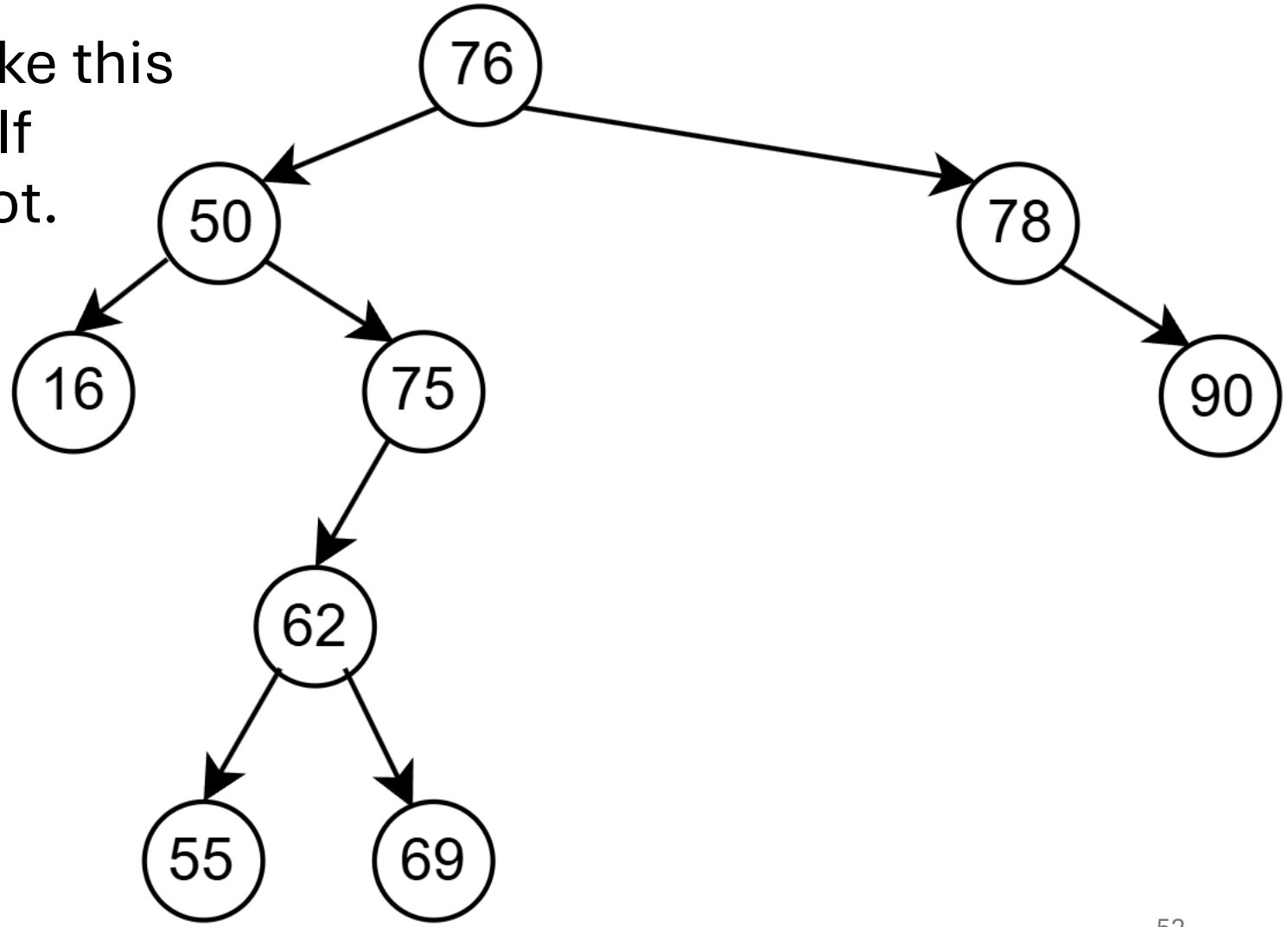
Splay Tree: A Self-Adjusting BST

- Let’s do an example! Take this BST, and search for 76. If found, splay it to the root.
- Done with the Zig-Zig, but still not at the root. What movement would be next?
- Zig-Zag: A DL rotation in this case.



Splay Tree: A Self-Adjusting BST

- Let’s do an example! Take this BST, and search for 76. If found, splay it to the root.
- Done!



Splay Tree: A Self-Adjusting BST

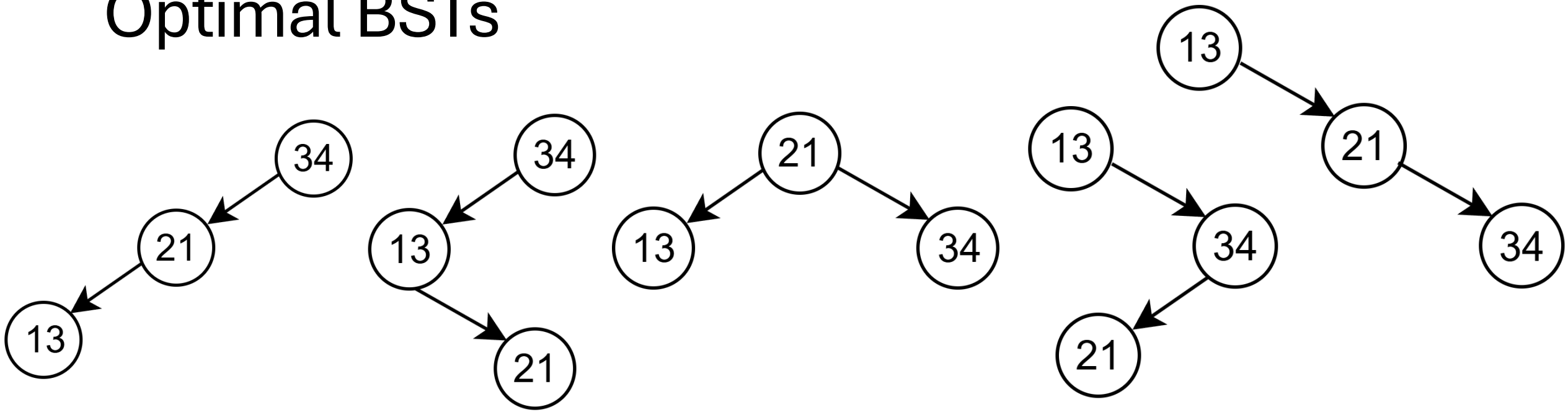
- Splay Trees are BSTs, but are they balanced BSTs?
- No! Unless it might be balanced from random chance, but the rotations aren’t for the purpose of balancing, just for moving the inserted (or search for) node to the root.

Splay Tree: A Self-Adjusting BST

- How are deletions handled for Splay Trees? There are many possible implementations, here is one of them:

1. Search for the victim, bringing it to the root.
2. Splay the victim's inorder predecessor to the root of the victim's left subtree.
3. Make the victim's right subtree be the inorder predecessor's right subtree.
4. Remove the victim's node (leaving the former inorder predecessor to be the tree's root)

Optimal BSTs



- There are 5 possible BSTs of three nodes, shown above.
- Which is the best one?
- Well, it depends: we could consider height, or ‘search popularity’
- We could compute average search times for each node, and structure the tree accordingly
- There’s a lot more to optimal BST story… but not in this course.